Porting SPaSM to Roadrunner

Roadrunner Technical Seminar Series

April 10, 2008

Sriram Swaminarayan

CCS-2 Roadrunner Applications Team

Acknowledgements

Folks that have wandered with me on this journey:

Tim Germann

Kai Kadau

Al McPherson

Nehal Desai

Special Thanks to Cornell Wright

Outline of This Talk

I: An Evolutionary Approach:
Morphing SPaSM onto Roadrunner for the Assessment

- Some SPU optimizations
- Mistakes we made
- Lessons we learned

II: A Revolutionary Approach

- Ripping SPaSM Apart
- Results for Lennard-Jones Potential

III: Odds and Ends

SPaSM is a Parallel, Scalable and Flexible MD Code

***S**calable **Pa**rallel **S**hort-range **M**olecular-dynamics*

- Shown to scale linearly to 212k CPUs on BG/L
  - Modeling of SPaSM by the PAL team shows this trend applies to Roadrunner as well
- Solves Newton's laws at the atomic level
- Aimed primarily at metals
- Modular
  - libSPaSM:
    Geometry, Communications,
    Visualization, and Iterators
  - libUser:
    Particle structure, Force Routines,
    Time-stepper, and Problem Specific Analysis

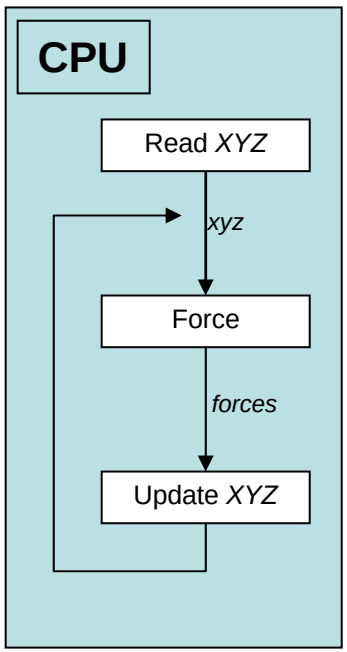
Part I: An Evolutionary Approach

Question:

Can we get acceleration with an evolutionary approach?

Decision to Implement force() on CBE

SPaSM Control/Data Flow



- 95% of time in force()
- Function offload model
- Opterons for positions
- CBE for force()
- No Problem!

CellMD: the CBE version concentrates on EAM

- Stand-alone on CBE
- First version of CellMD handles EAM (Embedded Atom Method) potential

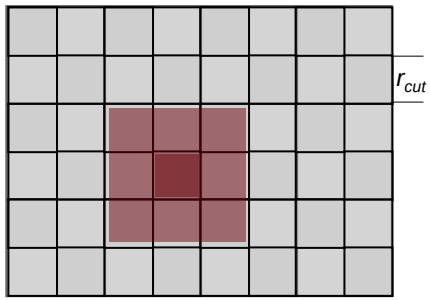
$$E_{sys} = \frac{1}{2}\sum_i \sum_{k \neq i} \phi(r_{ik}) + \sum_i F(\bar{\rho}_i) \quad where \quad \bar{\rho}_i = \sum_{k \neq i} \rho(r_{ik})$$

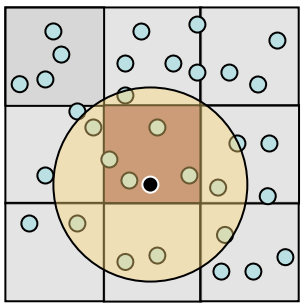
$$F_{ij} = \frac{\partial E_{sys}}{\partial r_{ij}} = \frac{\partial \phi_{ij}}{\partial r_{ij}} + \frac{\partial \rho}{\partial r_{ij}} \left(\left. \frac{\partial F}{\partial \bar{\rho}} \right|_{\bar{\rho}_i} + \left. \frac{\partial F}{\partial \bar{\rho}} \right|_{\bar{\rho}_j} \right)$$

$$= T_1 + T_2 \left(T_3^i + T_3^j \right)$$

  - 'Realistic' many Body potential
  - Uses tables for interpolation
  - Based on Density Functional Theory
  - Neighbor finding is the key to efficient implementation
- Correct before fast

Spatial Decomposition Key to Efficient Neighbor List





Typical Code:

for each box in system

load particles for box

load particles for neighbor boxes

Interact all particles in Box with all particles in all neighboring boxes

end

~ 20 atoms in each box

⇒ each atom interacts with 540 other atoms

⇒ However, only ~70 atoms lie within cutoff

⇒ Lots of wasted work

⇒ We need a means of rejecting atoms efficiently even within this reduced set

Optimizing the Neighbor Listing

Naive Method:
reject based on r_{ij}

```
for atoms i in current box {
   for atoms j in nbr boxes {
      compute r_ij
      if (r_ij < r_cut) {
        interact atoms i & j
      }
   }
}
```

Works OK on standard CPUs
Really sucks on the SPU

Simple NL:
Create a neighbor list first using r_{ij}

```
for atoms i in current box {
   for atoms j in nbr boxes {
      compute r_ij
      if (r_ij < r_cut) {
        add atom j to NL
        increment NL count
      }
   }
   { Process Neighbor List i }
}
```

Works very well on standard CPUs
OK on the SPU

Mutant SPU Version:
No 'if' statements

```
for atoms i in current box {
   for atoms j in nbr boxes {
      compute r_ij
      Mask = (r_ij > r_cut)
      Assign j to current NL position
      Increment NL pointer by (Mask & 1)
   }

   { Process Neighbor List i }
}
```

N/A on standard CPUs
Great on the SPU!

Method	Naive	Simple	Mutant
% time in NL	80%	60%	25%

SPU Code For Mutant Neighbor Listing

```
#define ZEROS ((vec_int4) {0,0,0,0})
#define ONES ((vec_int4) {1,1,1,1})

n = 0;
NL_Position = ZEROS;
for(j=0; j<numNbrAtoms; j++) {

    Δx = spu_sub(v_Ax[0],v_Bx[j]);
       Δy = spu_sub(v_Ay[0],v_By[j]);
       Δz = spu_sub(v_Az[0],v_Bz[j]);

   tmp1 = spu_mul(Δx, Δx);
   tmp2 = spu_madd(Δy,Δy,tmp1);
   r_sqr = spu_madd(Δz,Δz,tmp2);

    mask   = spu_cmpgt(r_sqr,rcut);
       mask_r = spu_slqwbyte(mask,8);
       incr   = spu_and(ONES,spu_or(mask, mask_r));

     NL_Δx[n] = Δx;
       NL_Δy[n] = Δy;
       NL_Δz[n] = Δz;
       NL_r2[n] = r_sqr;
       NL_Mask[n] = mask;

       NL_Position = spu_add(NL_Position,incr);
       n = si_to_int(NL_Position );
}
```

Mutant SPU Version:

Create a neighbor list first using r_{ij}

No 'if' statements

```
for atoms i in current box {
   for atoms j in nbr boxes {
      compute r_ij
      Mask = (r_ij > r_cut)
      Assign j to current NL position
      Increment NL pointer by (Mask & 1)
   }

   { Process Neighbor List i }
}
```

CellMD: Summary of Salient Features

- *3.5* × faster than SPaSM Opteron version
- ~ 6 Gflop/s double precision
- Forces computed using EAM potential
- Optimized Vector code
- Mutant neighbor listing
- Implements full neighbor list

  ⇒ Double the work of the CPU Version

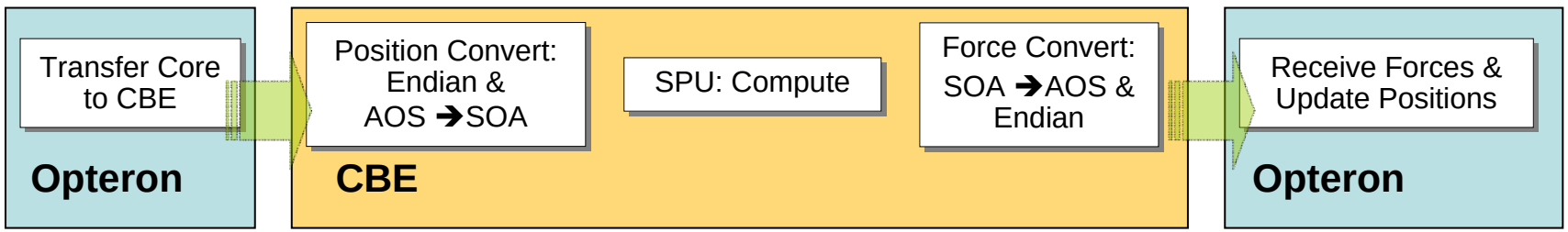
Ready to be merged with serial SPaSM code

Problem: CPU Data Layout Not Optimal for CBE

- SPaSM uses an Array of Structures (AOS) data layout:

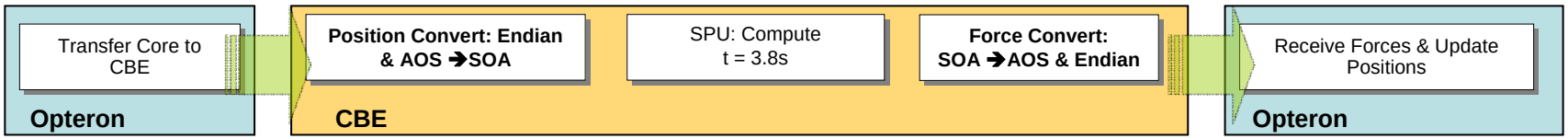
  $x_0 y_0 z_0 fx_0 fy_0 fz_0 \;\; x_1 y_1 z_1 fx_1 fy_1 fz_1 \ldots$
- CBE most effective with a Structure of Arrays (SOA) data layout:

  $x_0 x_1 x_2 \ldots x_N \;\; y_0 y_1 y_2 \ldots y_N \ldots\ldots$
- Also Endianness is different!
- Solution: Use PPU to translate Endian, and (AOS) to (SOA)

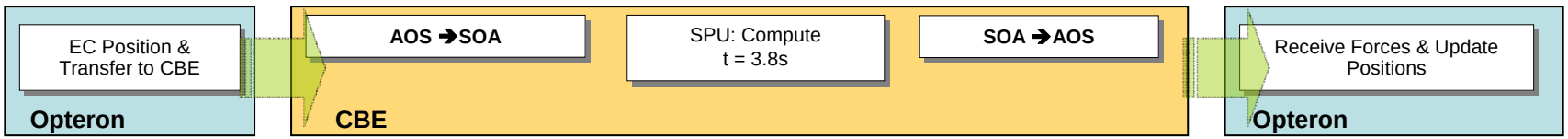


Mistake: The PPU is a Piece of Scrap

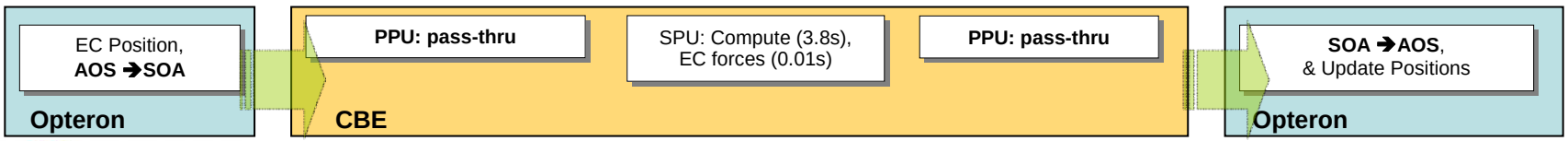
Worst: PPU does all conversions: t = 7.26, (t_{ovrhd}= *3.31*)



Not so bad: Opteron & PPU share Responsibility: t = 5.16, (t_{ovrhd}= *1.23*)

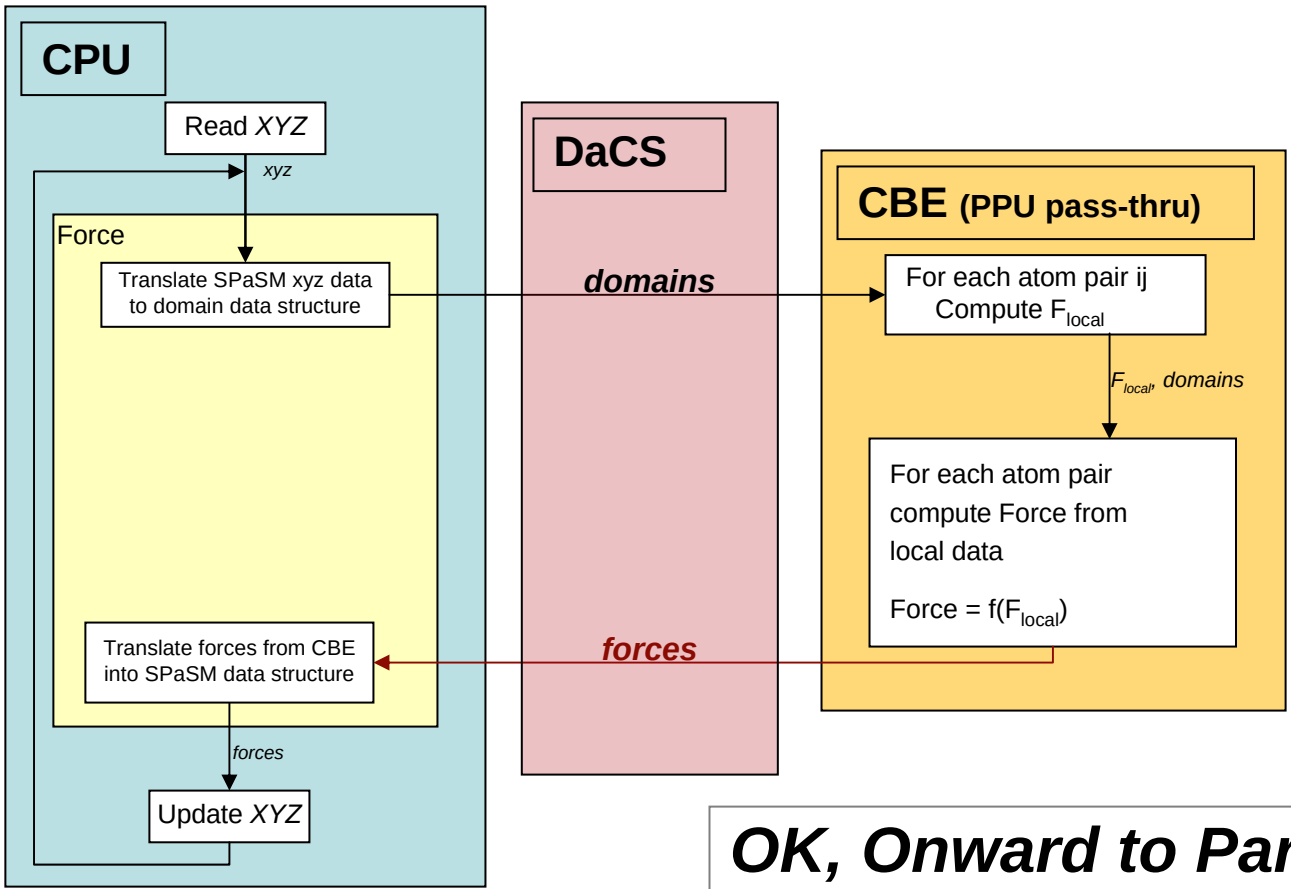


Best: Ignore PPU, Opteron & SPU share Responsibility: t = 4.3, (t_{ovrhd}= *0.5*)



Force Function Successfully Offloaded to CBE

2.5 × faster than Original Opteron Version

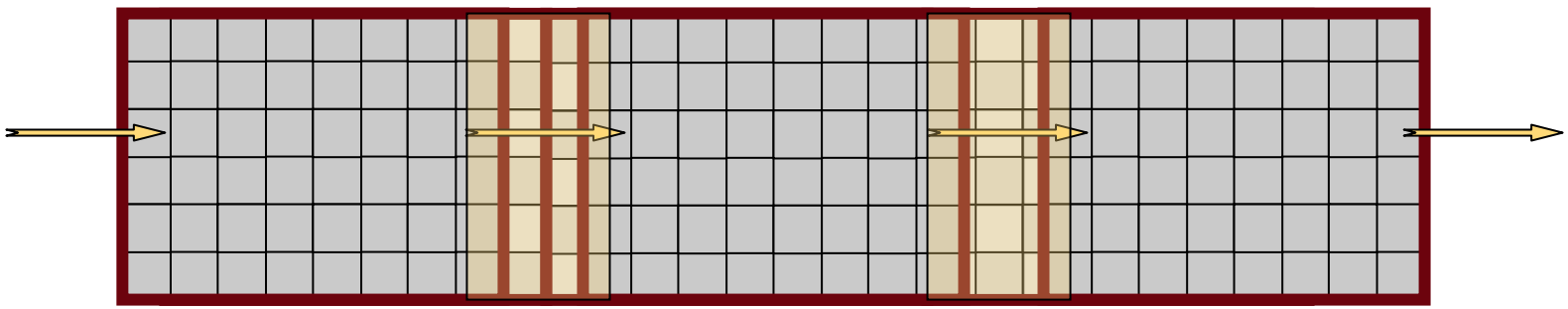


OK, Onward to Parallel!

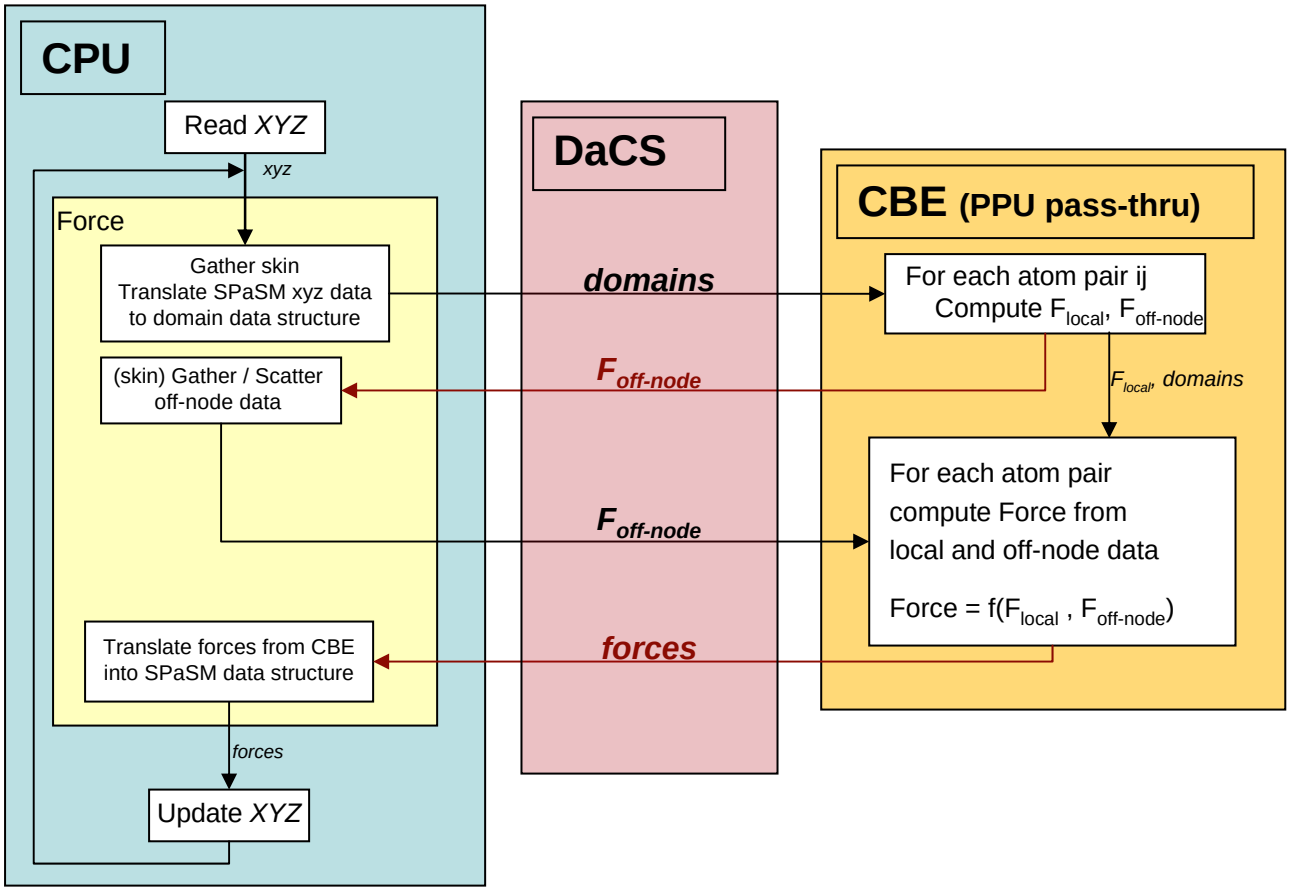
Problem: communications need to be redesigned

- SPaSM uses lock-step model
- Up to ***50,000*** messages per processor per timestep!
- Reduces memory requirements but …
- Kills performance in hybrid mode

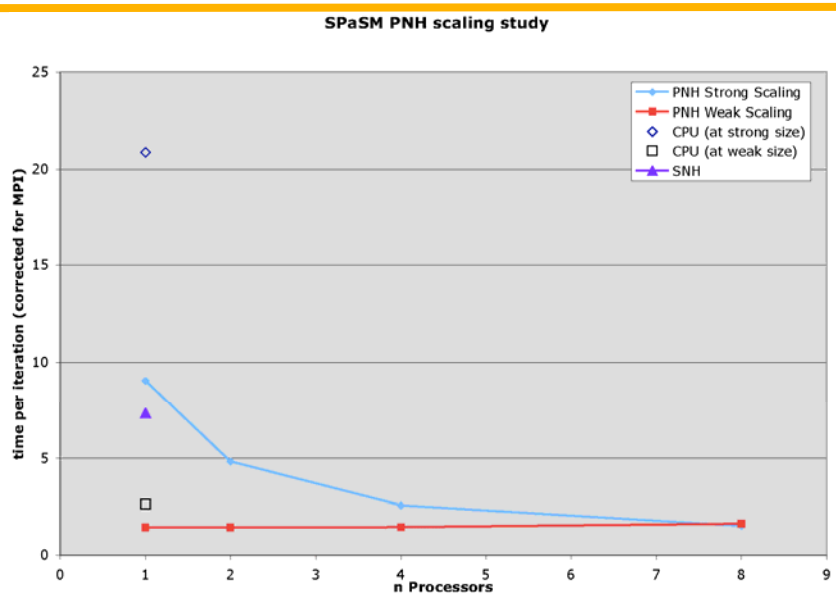
Solution: Use a ‘skin’ based model: drops to ***6*** messages



Parallel Hybrid Force Offload Works!



SPaSM-Parallel-Hybrid scales well on AAIS



- ~ 2× (3× on eDP) faster than SPaSM on Opterons
  - This is in spite of doing double the work
- 8 Gflop/s per Cell (100TF on Roadrunner)
- Opterons handle positions and MPI
- SPUs handle forces
- PPE does minimal memory management

Lessons Learned

- An evolutionary approach can work
- Use the SPUs and Opterons, avoid PPU.
  Remember: PPU = Poorly Performing Unit
- Memory layout and data flow is everything
- Endian conversion takes no time at all
- SIMD Vectorization is a must:
  - vector code is ~ 4 x faster than serial code
- ‘if’ statements on the SPU are a bad thing
  - Often doing the work twice and masking is better than using an ‘if’ or ‘switch’ statement
- Unroll loops manually since compilers are not there yet
- Did I mention the PPU sucks?

Success: Evolutionary Approaches Can Work

Question:

Can we get acceleration with an evolutionary approach?

Answer

Yes but you spend 20% of time translating data!

Lots of dead time on the compute units

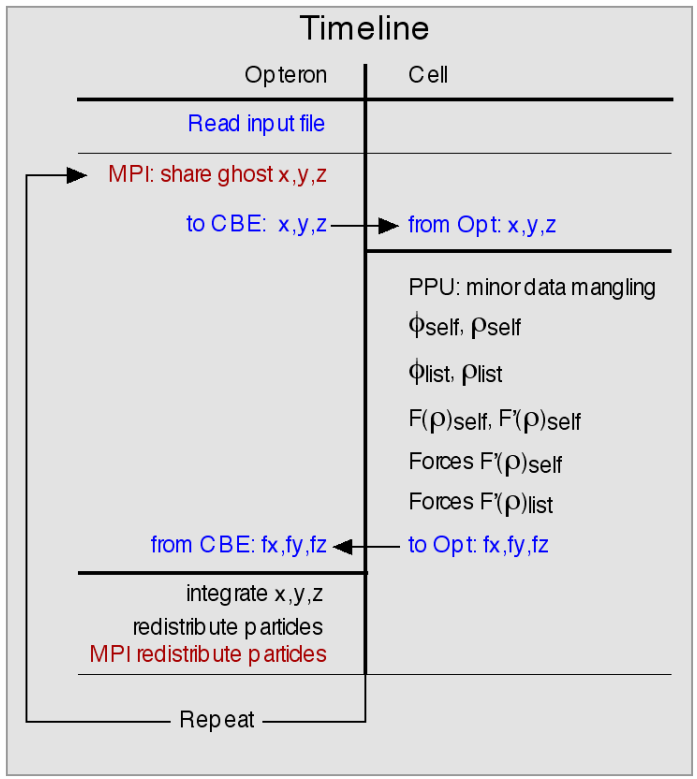
Part II: A Revolutionary Approach

Question:

Can we get much better acceleration by ripping SPaSM apart and rebuilding it starting from the CBE and working our way backwards, and is it worth it?

Timing Analysis of Part I Shows Dead Time

Opterons compute positions, then sit idle as the SPUs compute forces, and vice-versa



Self = interact within one box
List = interact between boxes

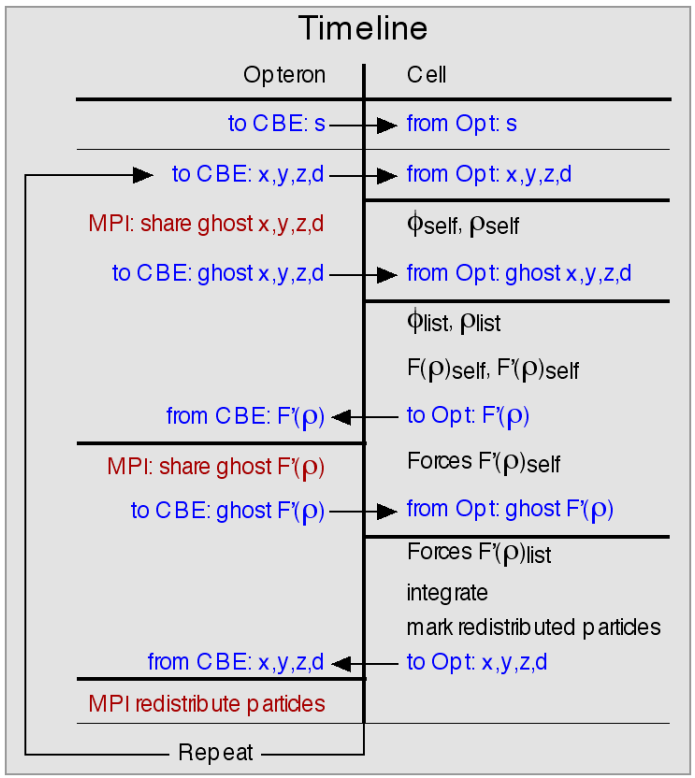
$$E_{sys} = \frac{1}{2}\sum_i \sum_{k \neq i} \phi(r_{ik}) + \sum_i F(\bar{\rho}_i) \quad where \quad \bar{\rho}_i = \sum_{k \neq i} \rho(r_{ik})$$

$$F_{ij} = \frac{\partial E_{sys}}{\partial r_{ij}} = \frac{\partial \phi_{ij}}{\partial r_{ij}} + \frac{\partial \rho}{\partial r_{ij}} \left(\frac{\partial F}{\partial \bar{\rho}}\bigg|_{\bar{\rho}_i} + \frac{\partial F}{\partial \bar{\rho}}\bigg|_{\bar{\rho}_j} \right)$$

$$= T_1 + T_2 \left(T_3^i + T_3^j \right)$$

New Structure: SPU is King, Communications Hidden

Opterons handle communications with other nodes.
SPUs do everything else



Self = interact within one box
List = interact between boxes

$$E_{sys} = \frac{1}{2}\sum_i \sum_{k \neq i} \phi(r_{ik}) + \sum_i F(\bar{\rho}_i) \quad where \quad \bar{\rho}_i = \sum_{k \neq i} \rho(r_{ik})$$

$$F_{ij} = \frac{\partial E_{sys}}{\partial r_{ij}} = \frac{\partial \phi_{ij}}{\partial r_{ij}} + \frac{\partial \rho}{\partial r_{ij}} \left(\frac{\partial F}{\partial \bar{\rho}}\bigg|_{\bar{\rho}_i} + \frac{\partial F}{\partial \bar{\rho}}\bigg|_{\bar{\rho}_j} \right)$$

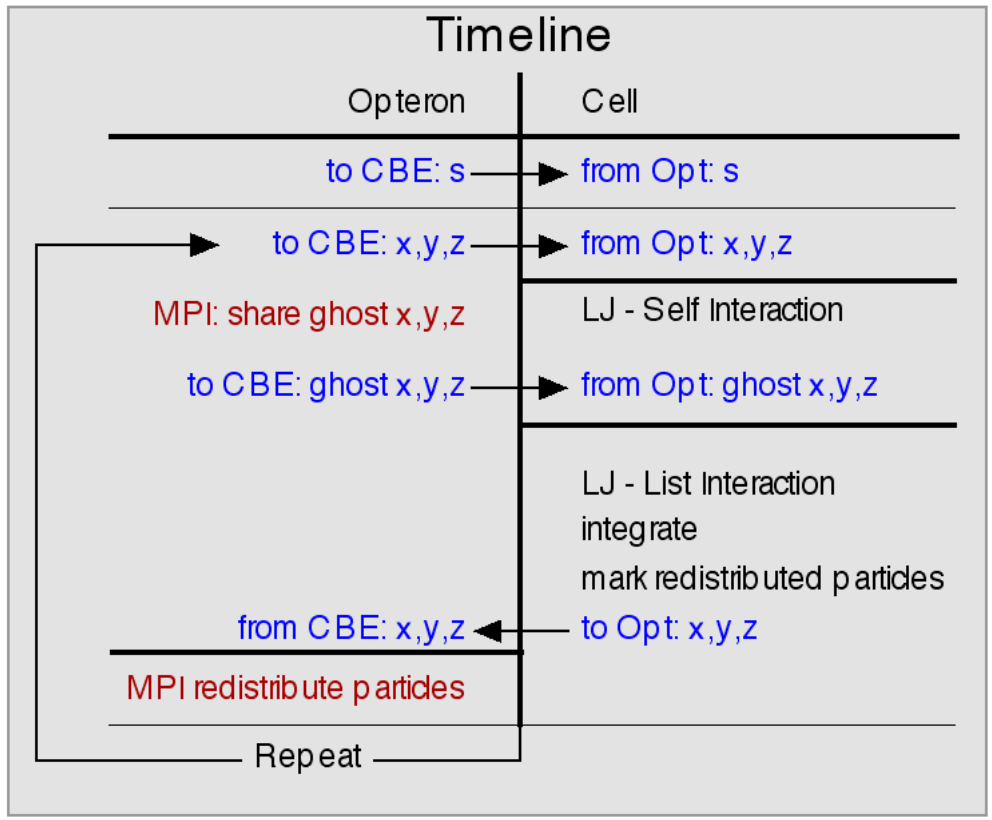
$$= T_1 + T_2 \left(T_3^i + T_3^j \right)$$

Simplified flow for Lennard-Jones Potential

Opteron:
- Own all Off-node communication
- Lots of dead time - used for analysis

SPU:
- Own all Compute intensive parts
- Very little dead time

Added benefits:
- Simpler potential
- More opportunities to optimize SPU code



Self = interact within one box
List = interact between boxes

Serial Results Show Promising Performance

Opteron:

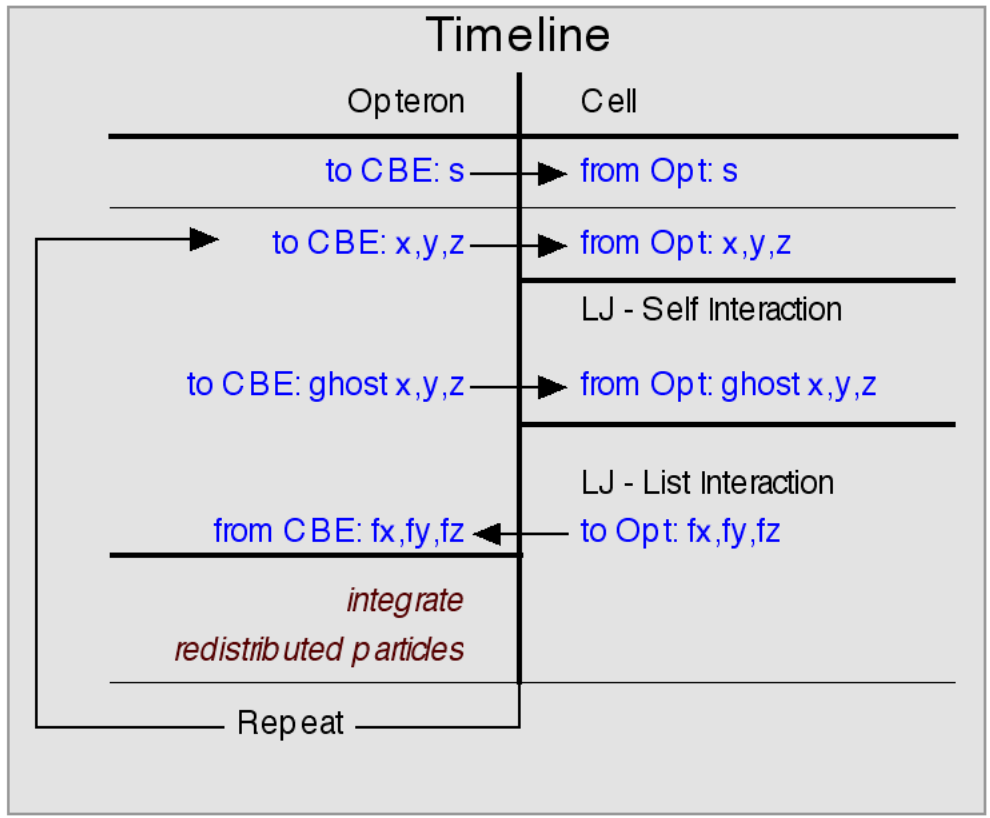
- Integrate time
- Distribute particles

SPU:

- Computes forces

Preliminary Results:

- ~ 6× faster than SPaSM on the base Opterons
- Kernel runs at 45% of peak
- 28 Gflop/s overall performance
- Projected 300 TF on Roadrunner in double precision!



Self = interact within one box
List = interact between boxes

Thoughts on DaCS

- For the most part, DaCS just works
- Use dacs_put() and dacs_get() if you can
- If you are transitioning from MPI, you will fight the good fight with DaCS
  - dacs_send() and dacs_recv() have limitation on tags
  - No equivalent of MPI_IProbe()
  - dacs_recv() will not report message size or the tag of the received message
  - Need to know a-priori the sizes and order of messages
- New words in the English Language:

Dacsify, Dacsificate:	To convert a code to DaCs
Dacs-phyxiate:	How you feel when doing said conversion
Dacs-ygen:	What your code needs to live. Lack of dacsygen cause dacs-phyxiation

Part II: Philosophical Question

Is it still SPaSM if you rip its guts out and rebuild it from the ground up?

Part III: Odds & Ends

- Using Roadrunner efficiently is hard
- We've developed some tools to make Roadrunner more 'usable' for us
- The community is welcome to them
- They come with absolutely no support whatsoever
- Nor do they come with any warranty
- Documentation?
  - Hah! I laugh in the face of documentation

SAL: SIMD Abstraction Layer

Makes SIMD instructions universal across SPU, SSE, and AltiVec

Authors: Ben Bergen & Tim Kelley

What it provides:

- Single API for vectorization
- Commands expand to corresponding platform's underlying SIMD vector units

What it doesn't provide:

- DMA engine
- Automatic vectorization

CIK: Cell Isolation Kit

Allows for debugging of data flow within SPU code on Opteron or PPU

Author: Sriram Swaminarayan

What it provides:

- Fake DMA Engine
- SIMD abstraction (will soon switch to SAL)
- Same source compiles and runs on the Opterons, PPU, and SPUs

What it doesn't provide:

- Automatic vectorization

Mutant Neighbor Listing with CIK

```
#define ZEROS ((vec_int4) {0,0,0,0})
#define ONES ((vec_int4) {1,1,1,1})

n = 0;
NL_Position = ZEROS;
for(j=0; j<numNbrAtoms; j++) {

    Δx = spu_sub(v_Ax[0],v_Bx[j]);
       Δy = spu_sub(v_Ay[0],v_By[j]);
       Δz = spu_sub(v_Az[0],v_Bz[j]);

   tmp1 = spu_mul(Δx, Δx);
   tmp2 = spu_madd(Δy,Δy,tmp1);
   r_sqr = spu_madd(Δz,Δz,tmp2);

    mask   = spu_cmpgt(r_sqr,rcut);
       mask_r = spu_rlqwbyte(mask,8);
       incr   = spu_and(ONES,spu_or(mask, mask_r));

     NL_Δx[n] = Δx;
       NL_Δy[n] = Δy;
       NL_Δz[n] = Δz;
       NL_r2[n] = r_sqr;
       NL_Mask[n] = mask;

       NL_Position = spu_add(NL_Position,incr);
       n = si_to_int(NL_Position );
}
```

```
#define ZEROS ((cik32i_t) {0,0,0,0})
#define ONES ((cik32i_t) {1,1,1,1})

n = 0;
NL_Position = ZEROS;
for(j=0; j<numNbrAtoms; j++) {

    Δx = cikSub32fp(v_Ax[0],v_Bx[j]);
       Δy = cikSub32fp(v_Ay[0],v_By[j]);
       Δz = cikSub32fp(v_Az[0],v_Bz[j]);

   tmp1 = cikMul32fp(Δx, Δx);
   tmp2 = cikMAdd32fp(Δy,Δy,tmp1);
   r_sqr = cikMAdd32fp(Δz,Δz,tmp2);

    mask   = cikCmpgt32fp(r_sqr,rcut);
       mask_r = cikRotate32(mask,8);
       incr   = cikAnd32i(ONES,cikOr32i(mask, mask_r));

     NL_Δx[n] = Δx;
       NL_Δy[n] = Δy;
       NL_Δz[n] = Δz;
       NL_r2[n] = r_sqr;
       NL_Mask[n] = mask;

       NL_Position = cikAdd32i(NL_Position,incr);
       n = cikToInt32i(NL_Position );
}
```

Loop Unroller

Unrolls loops based on user input

Author: Sriram Swaminarayan

What it provides:

- Automation of loop unrolling to any depth
- Additive collation of unrolled variables at the end

What it doesn't provide:

Code analysis to see if what you asked it to unroll actually can be unrolled 'safely'. Essentially YAFIYGI.

Importance of Loop Unrolling

For the simple code:

```
vec_double2 a[8000];
vec_double2 e_res
init(a);
e_res = {0.0,0.0};
// e_res = sum(a[i]^2)
for(int i=0;i<8000;i++) {
   e_res = spu_madd(a[i],a[i],e_res);
}
```

Unroll Depth	xlc	gcc
0	11.4 GF	10.9 GF
2	22.4 GF	21.2 GF
4	44.7 GF	31.4 GF
8	48.2 GF	30.3 GF
16	87.1 GF	49.1 GF

```
/*--------------  Unrolled by 16  ------------------------*/
  res0 = res1 = res2 = res3 = spu_splats((double)0.0);
  res4 = res5 = res6 = res7 = spu_splats((double)0.0);
  res8 = res9 = res10 = res11 = spu_splats((double)0.0);
  res12 = res13 = res14 = res15 = spu_splats((double)0.0);
for(i=0; i<NMAX; i+=16) {
   res0 = spu_madd(a[i],a[i],res0);
   res1 = spu_madd(a[i+1],a[I+1],res1);
   res2 = spu_madd(a[i+2],a[i+2],res2);
   res3 = spu_madd(a[i+3],a[i+3],res3);
   res4 = spu_madd(a[i+4],a[i+4],res4);
   res5 = spu_madd(a[i+5],a[i+5],res5);
   res6 = spu_madd(a[i+6],a[i+6],res6);
   res7 = spu_madd(a[i+7],a[i+7],res7);
   res8 = spu_madd(a[i+8],a[i+8],res8);
   res9 = spu_madd(a[i+9],a[i+9],res9);
   res10 = spu_madd(a[i+10],a[i+10],res10);
   res11 = spu_madd(a[i+11],a[i+11],res11);
   res12 = spu_madd(a[i+12],a[i+12],res12);
   res13 = spu_madd(a[i+13],a[i+13],res13);
   res14 = spu_madd(a[i+14],a[i+14],res14);
   res15 = spu_madd(a[i+15],a[i+15],res15);
 }
  res8 = spu_add(res8,res9);
  res10 = spu_add(res10,res11);
  res12 = spu_add(res12,res13);
  res14 = spu_add(res14,res15);
  res9 = spu_add(res8,res10);
  res11 = spu_add(res12,res14);
  res8 = spu_add(res11,res9);
  res0 = spu_add(res0,res1);
  res2 = spu_add(res2,res3);
  res4 = spu_add(res4,res5);
  res6 = spu_add(res6,res7);
  res1 = spu_add(res0,res2);
  res3 = spu_add(res4,res6);
  res5 = spu_add(res3,res1);
  e_res = spu_add(res5,res8);
```

Parts I, II, & III: Deep Thoughts

Before You Begin Programming Roadrunner:

Analyze your application's performance

- Profile your code to find hot spots
- Vectorization is essential to performance: Analyze hot spots to see if they can be vectorized
- Data layout / data flow is everything:
  - If you cannot vectorize without modifying your data, you are already dead
  - Consider a revolutionary approach
- Overlap compute and communications
- Manually unroll all possible loops
- Avoid 'if's if you can
- PAL team can help you do projections to large scales based on your communication patterns
- ***Keep the SPUs busy!***